GD2S03
Advanced Software Engineering & Programming for Games

MEDIA DESIGN SCHOOL

Bachelor of Software Engineering(BSE)
Game Development

- Overview
  - ➢The Basics
  - ➢Basic Operators
  - ➢Strings and Characters
  - ➢Collection Types
  - ➢Control Flow

The Basics – Constants, Variables, Comments and Semicolon

- Declaring constant – using ‘let’ keyword
- Declaring variable – using ‘var’ keyword

```
let constNumber = 10
var varNumber = 0
```

- Multiple variables or constants can be declared in single line, separated by comma

```
var x = 0.0, y = 0.0, z = 0.0
```

- **Type Annotation** makes clear the kind of values a constants or variable can store.
- Multiple variable of same type can be defined in single line

```
var anyString: String
var anyInt: Int
var red, green, blue: Double
```

- Constants and variables name can contain almost any character, including Unicode characters.

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
let schoolName = "MDS"
```

- Values can be printed using “print(_:separator:terminator:)” function

```
print(schoolName)
print(" schoolName is \(schoolName)")
```

- Single line comment – two forward slash
- Multiple line comments - /* comment goes here */

```
// This is a comment.
/* This is also a comment
   but is written over multiple lines.
*/
```

- Semicolon at the send of statement is optional

The Basics – Integer, Floating-Points, Boolean and Type Aliases

To Access the minimum value of an integer - min
To Access the maximum value of an integer - max

```
let minVal = UInt8.min
let maxVal = UInt8.max
```

On 32 bit platform – Int is same as Int 32
On 32 bit platform – Uint is same as UInt32

Double represents 64 bit floating point number
Float represents 32 bit floating point number

Swift provides two Boolean constant - ***true and false***

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

Integer conversion is explicit. This opt-in approach prevents hidden conversion error.

```
let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
let twoThousandAndOne = twoThousand + UInt16(one)
```

Conversion between integer and floating point numeric types must be made explicit.

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
```

Type Aliases define an alternative name for an existing type. Type Aliases can be defined with *typealias* keyword

```
typealias AudioSample = UInt16
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound is now 0
```

The Basics – Type Safe, Type Inference and Numeric Literals

Swift is **type safe** language, meaning an *Int* cannot be passed where a *string* is required. In case of type mismatched an error is flagged.

```
var name: String = 10 // type mismatch error
```

Type Inference – The type of the variable is inferred and lesser type declaration is required.

```
let meaningOfLife = 42
// meaningOfLife is inferred to be of type Int
```

Integer literals can be written as:

- •A *decimal* number, with no prefix
- •A *binary* number, with a 0b prefix
- •An *octal* number, with a 0o prefix
- •A *hexadecimal* number, with a 0x prefix

```
let decimalInteger = 17
let binaryInteger = 0b10001 // 17 in binary notation
let octalInteger = 0o21 // 17 in octal notation
let hexadecimalInteger = 0x11 // 17 in hexadecimal
notation
= 0x11 // 17 in hexadecimal notation
```

The Basics – Tuples

Tuples group multiple values into a single compound value.
A tuple can be made for any permutation - (Int, Int, Int) or (String, Bool)

```
let http404Error = (404, "Not Found")
/* http404Error is of type (Int, String), and equals
(404, "Not Found") */
```

Tupule contents can be decomposed into separate constants or variables.

```
let (statusCode, statusMessage) = http404Error
print("The status code is \(statusCode)")
// Prints "The status code is 404"
```

To get only some part of the tuple, other parts can be ignored by using underscore(_)

```
let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")
// Prints "The status code is 404"
```

Individuals elements can be accessed using indices starting at zero

```
print("The status code is \(http404Error.0)")
// Prints "The status code is 404"
```

Individual elements can be named at tupule definition.

```
let http200Status = (statusCode: 200, description: "OK")
print("The status code is \(http200Status.statusCode)")
// Prints "The status code is 200"
```

Tupules are particularly useful as the return values of a function

```
func returnATuple(input: Int)->(Int, String){
    return(input, String(input))
}
print(returnATuple(input: 55))
```

The Basics – Optionals

optionals are used in two situations

- Either there is no value
- There is a value and can be unwrapped.

Int optional - `Int?`

The question mark indicates that it might contain some int value, or it might contain no value at all.

The default value of an optional variable is nil

```
var surveyAnswer: String?
// surveyAnswer is automatically set to nil
```

An optional value can be set to valueless state by assigning it the *nil* value

```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int value of 404

serverResponseCode = nil
// serverResponseCode now contains no value
```

The Basics - Optionals

Forced Unwrapping -

exclamation mark(!) is used to unwrap the value from optional

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertedNumber is inferred to be of type "Int?", or "optional Int"
print("convertedNumber has an integer value of \(convertedNumber!).")
```

Optional Binding - to find out whether an optional contains a value, and if so, to make that value available as a temporary constant or variable.

```
if let actualNumber = Int(possibleNumber) {
    print("\"\(possibleNumber)\" has an integer value of \(actualNumber)")
} else {
    print("\"\(possibleNumber)\" could not be converted to an integer")
}
// Prints ""123" has an integer value of 123"
```

Basic Operators

Arithmetic Operator (+, -, *, /)	`let a = 9 + 4 //[a = 13]`
Remainder Operator (%)	`let a = 9 % 4 //[a = 1]`
Compound Assignment Operators(+=, -= … etc)	`var a = 1 ; a += 2 //[a = 3]`

Comparison Operator. (==, !=, >, <, >=, <=)
identity operators (=== and !==), two object references to the same object instance or not
	`a > b //[true if a is greater than b]`
	`a == b //[true if a is equal to b]`
	`a != b //[true if a is not equal to b]`

Ternary conditional Operator (? :)	`c = a > b ? a : b`
Nil-Coalescing Operator	`a != nil ? a! : b`

Logical Operators (NOT(!), OR(||), AND(&&))	`let a = true, b = false`
	`!a [false], a && b [false], a || b //[true]`

Closed Range Operators (a…b) - define range from 'a' to 'b' including 'a' and 'b'	`for index in 1...5 {`
	`    print ("\(index) times 5 is \(index * 5)")}`

Half-Open range Operator(a..<b)- define range that run from 'a' to 'b' but doesn't include 'b'	`for index in 1..<5 {`
	`    print ("\(index) times 5 is \(index * 5)")}`

One-Sided Ranges - ranges that continue in one direction as far as possible	`1... [gives 1, 2, 3 ... end of elements]`
	`...2 [gives beginning to ...1,2]`

Strings and Characters

String literals. Value types	`let someString = "Some string literal value"`

Multiline String Literals	`let quotation = """`
	`The White Rabbit put on his spectacles`
	`"""`

Initializing an Empty String	`var emptyString = " " // empty string literal`
	`var anotherEmptyString = String() // initializer syntax`

String Mutability - whether a string can be modified (mutated) or not. Constant(let) strings cannot be mutated	`var variableString = "Horse"`
	`variableString += " and carriage"`
	`// variableString is now "Horse and carriage"`

Use Character type annotation to create a stand-alone Character constant or variable	`let exclamationMark: Character = "!"`
	`var catCharacters: [Character] = ["C", "a", "t", "!", "🐱"]`

Concatenating Strings	`let string1 = "hello", string2 = "there"`
	`var welcome = string1 + string2`

Concatenating Strings and Characters	`welcome.append(exclamationMark)`

String Interpolation	`let multiplier = 3`
	`let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)” 2.5)"`

Strings and Characters

Counting Characters (count)	`var word = "cafe"`
	`print("the number of characters in \(word) is \(word.count)")`

String Indices - String.Index, corresponds to the position of each Character in the string.
- startIndex, endIndex
- index(before:), index(after:)
- index(_:offsetBy:)
	`let greeting = "Guten Tag!"`
	`greeting[greeting.startIndex] //G`
	`greeting[greeting.index(before: greeting.endIndex)]   // !`
	`greeting[greeting.index(after: greeting.startIndex)]  // u`
	`let index = greeting.index(greeting.startIndex, offsetBy: 7)`
	`greeting[index] // a`

indices - access all of the indices of individual characters in a string.	`for index in greeting.indices {`
	`    print("\(greeting[index]) ", terminator: "") }`

Inserting
- insert(_:at:)
- insert(contentsOf:at:)
	`var welcome = "hello"`
	`welcome.insert("!", at: welcome.endIndex) //  "hello!"`
	`welcome.insert(contentsOf: " there", at: welcome.index(before: welcome.endIndex))`

Removing
- remove(at:)
- removeSubrange(_:)
	`welcome.remove(at: welcome.index(before:welcome.endIndex))`
	`// welcome now equals "hello there"`
	`let range = welcome.index(welcome.endIndex, offsetBy: –6)..<welcome.endIndex`
	`welcome.removeSubrange(range) // welcome now equals "hello"`

Strings and Characters

Substrings-

- substrings aren’t suitable for long-term storage—because they reuse the storage of the original string, the entire original string must be kept in memory as long as any of its substrings are being used.
- to store the result for a longer time, convert the substring to an instance of **String**

```
let greeting = "Hello, world!"
let index = greeting.index(of: ",") ??
greeting.endIndex
let beginning = greeting[..<index]
// beginning is "Hello"

// Convert the result to a String for long-term
storage.
let newString = String(beginning)
```

String Comparisons - Swift provides three ways to compare textual values:

- string and character equality (==, !=)
- prefix equality (hasPrefix)
- suffix equality(hasSuffix)

```
let string1 = "Hello", string2 = "Hello"
if string1 == string2{/* do something */}

let scene = "Act 1 Scene 1: Verona, A public place"
if scene.hasPrefix( "Act"){/* do something */}
if scene.hasSuffix("place"){/* do something */}
```

Unicode Representation of strings

1. UTF - 8
2. UTF-16
3. Unicode Scalar

```
let dogString = "Dog!!🐶"
for codeUnit in dogString.utf8 { }
for codeUnit in dogString.utf16 { }
for scalar in dogString.unicodeScalars { }
```

Collection Types

Arrays -
- stores values of same type.
- an ordered list
- Can be mutable and immutable
- Can contain default values
	`var someInts = [Int]() //empty array`
	`someInts = [] //now empty array`
	`var threeDoubles = Array(repeating:0.0, count:3)`
	`var anotherThreeDoubles = Array(repeating: 2.5, count: 3)`

• Two arrays can be added	`var sixDoubles = threeDoubles + anotherThreeDoubles`

• Creating an array with array literal	`var shoppingList: [String] = ["Eggs", "Milk"] //OR`
	`var shoppingList = ["Eggs", "Milk"]`

• Accessing and Modifying Array
- count
- isEmpty
- append (+=)
- subscript([])
- insert(_:at:)
- remove(at:)->Any gaps are closed
- removeLast()
	`print("The shopping list contains \(shoppingList.count) items.")`
	`if shoppingList.isEmpty {/* do something */}`
	`shoppingList.append("Flour");`
	`shoppingList += ["Baking Powder"]`
	`var firstItem = shoppingList[0];`
	`shoppingList[0] = "Six eggs"`
	`shoppingList[4...6] = ["Bananas", "Apples"] // replace 4,5,6`
	`shoppingList.insert("Maple Syrup", at: 0)`
	`let mapleSyrup = shoppingList.remove(at: 0)`
	`let apples = shoppingList.removeLast()`

• Iterating Over an Array	`for item in shoppingList { print(item) }`

Collection Types - Sets

- stores distinct values of same type
- no defined ordering

```
var letters = Set<Character>() // no shorthand notation like array
letters.insert("a")
letters = [] // empty set
```

- Creating a set with an array literal

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"] //OR
var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]
```

- Accessing and Modifying Array
  - Count -isEmpty
  - insert(_:) -remove(_:)
  - removeAll() -contains(_:)

```
print("I have \(favoriteGenres.count) favorite music genres.")
if favoriteGenres.isEmpty { /* do something */ }
favoriteGenres.insert("Jazz")
let removedGenre = favoriteGenres.remove("Rock")
if favoriteGenres.contains("Funk") { /* do something */ }
```

- Iterating Over a Set

```
for genre in favoriteGenres {print("\(genre) "}
```

- sorted() - to order the items

```
for genre in favoriteGenres.sorted() { /* do something */}
```

- Performing Set Operations
- intersection
- symmetricDifference
- Union
- subtracting

```
//Let a and b are two sets
a.intersection(b)
a.symmetricDifference(b)
a.Union(b)
a.Subtracting(b)
```

Collection Types - Dictionary

- Key(unique) value pair
- no specific ordering

```
var namesOfIntegers = [Int: String]() // empty Dictionary
namesOfIntegers[16] = "sixteen" //one key value pair
namesOfIntegers = [:]  //once again empty
```

- Creating a Dictionary with a Dictionary literal

```
var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB":
"Dublin"]  OR
var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

- Accessing and Modifying A Dictionary
- count
- isEmpty
- updateValue(_:forKey:)
- removeValue(forKey:)

```
airports["LHR"] = "London" //new key and value inserted
airports["LHR"] = "London Heathrow" //value updated
print("The airports dictionary contains \(airports.count) items.")
if airports.isEmpty {  /* do something */ }
let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB")
             OR
let airportName = airports["DUB"]
airports["APL"] = nil // APL removed
let removedValue = airports.removeValue(forKey: "DUB")
                   // DUB removed and returned
```

- Iterating Over a Dictionary

```
for airportCode in airports.keys{ print("\(airportCode)")}
for airportName in airports.values { print("\(airportName)")}
```

- Initialize a new array with the keys or values property

```
let airportCodes = [String](airports.keys) // airportCodes is
["YYZ", "LHR"]
let airportNames = [String](airports.values)
// airportNames is ["Toronto Pearson", "London Heathrow"]
```

Control Flow - Loop

- for-in
- to iterate over a sequence

```
for index in 1...5 { print("\(index) times 5 is \(index * 5)")}
for tickMark in 0..<60 { /* render the tick mark each minute
(60 times)*/}
for tickMark in stride(from: 0, to: 60, by: 5) {/* render the
tick mark every 5 minutes (0, 5, 10, 15 ... 45, 50, 55)*/}
for tickMark in stride(from: 3, through: 12, by: 3) {/* render
the tick mark every 3 hours (3, 6, 9, 12)*/ }

let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}
let base = 3, power = 10
for _ in 1...power { answer *= base }
```

- While
- evaluates its condition at the start of each pass through the loop.

```
let finalSquare = 25, square = 0
while square < finalSquare {
    //do something
}
```

- Repeat-while
- evaluates its condition at the end of each pass through the loop.

```
repeat{
    //do something
} while(square < finalSquare)
```

Control Flow – if-else, switch

- if

```swift
var temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
```

- Switch

```
switch some value to consider {
case value 1:
    respond to value 1
case value 2,value 3:
    respond to value 2 or 3
default:
    otherwise, do something else
}
```

```swift
let someCharacter: Character = "z"
switch someCharacter {
case "a":
    print("The first letter of the alphabet")
case "z":
    print("The last letter of the alphabet")
default:
    print("Some other character")
}
// Prints "The last letter of the alphabet"
```

Control Flow - Switch

- No implicit Fallthrough - the entire switch statement finishes its execution as soon as the first matching switch case is completed, without requiring an explicit break statement.

- The body of each case must contain at least one executable statement.
- It is not valid to write the following code, because the first case is empty:

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a": // Invalid, the case has an empty body
case "A":
    print("The letter A")
default:
    print("Not the letter A") // This will report a compile–
time error.
}
```

- **Compound case**
- To make a switch with a single case that matches both "a" and "A", combine the two values into a compound case, separating the values with commas.

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a", "A":
    print("The letter A")
default:
    print("Not the letter A")   // Prints "The letter A"
}
```

Control Flow - Switch

- Interval matching

```swift
let approximateCount = 62
let naturalCount: String
let countedThings = "moons orbiting Saturn"


switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
case 12..<100:
    naturalCount = "dozens of"
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}
print("There are \(naturalCount) \(countedThings).")
// Prints "There are dozens of moons orbiting Saturn."
```

Control Flow - Switch

- Switch – Interval matching – Tuples

- Swift allows multiple switch cases to consider the same value or values. In fact, the point (0, 0) could match all *four* of the cases in this example.

- However, if multiple matches are possible, the first matching case is always used.

- The point (0, 0) would match case (0, 0) first, and so all other matching cases would be ignored.

```swift
let somePoint = (1, 1)


switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the box")
default:
    print("\(somePoint) is outside of the box")
}
// Prints "(1, 1) is inside the box"
```

Control Flow - switch

- Switch – Interval matching – value bindings

- A switch case can name the value or values it matches to temporary constants or variables, for use in the body of the case.
- This behavior is known as *value binding*, because the values are bound to temporary constants or variables within the case’s body.

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    print("on the x-axis with an x value of \(x)")
case (0, let y):
    print("on the y-axis with a y value of \(y)")
case let (x, y):
    print("somewhere else at (\(x), \(y))")
}// Prints "on the x-axis with an x value of 2"
```

- Switch – where
- A switch case can use a *where* clause to check for additional conditions.

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
    print("(\(x), \(y)) is on the line x == y")
case let (x, y) where x == -y:
    print("(\(x), \(y)) is on the line x == -y")
case let (x, y):
    print("(\(x), \(y)) is just some arbitrary point")
}
// Prints "(1, -1) is on the line x == -y"
```

Control Flow - switch

- Switch – compound cases

- Multiple switch cases that share the same body can be combined by writing several patterns after case, with a comma between each of the patterns.
- If any of the patterns match, then the case is considered to match. The patterns can be written over multiple lines if the list is long

```
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
     "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
    print("\(someCharacter) is a consonant")
default:
    print("\(someCharacter) is not a vowel or a consonant")
}
// Prints "e is a vowel"
```

- Switch – compound cases – value binding
- Compound cases can also include value bindings.

```
let stillAnotherPoint = (9, 0)
switch stillAnotherPoint {
case (let distance, 0), (0, let distance):
    print("On an axis, \(distance) from the origin")
default:
    print("Not on an axis")
}
// Prints "On an axis, 9 from the origin"
```

Control Flow – Control Transfer Statements

- *continue*
- stop everything and start again at the beginning of the next iteration through the loop.

- *break*
- ends execution of an entire control flow statement immediately.

- *Labeled statements*
- used to be explicit about which loop or conditional statements should be terminated using ‘break’.
- If the break statement above did not use the *gameLoop* label, it would break out of the switch statement, not the while statement.
- *gameLoop* label makes clear which control statement should be terminated.

```
var number: UInt32 = 0

gameLoop: while number >= 0 {
    number = arc4random_uniform(15)
    switch(number){

    case let x where x % 3 == 0:
        print(x)

    case let x where x % 4 == 0:
        print(number)

    case let x where x % 7 == 0:
        print(number)
        break gameLoop

    default:
       break
    }
}
```

Control Flow – Control Transfer Statements

- ***fallthrough***
  - In Swift, switch statements don’t fall through the bottom of each case and into the next one.

  - But this behavior can be obtained on a case-by-case basis with the fallthrough keyword

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
// Prints "The number 5 is a prime number, and also an
integer."
```

- ***Checking API Availability***
  - Swift has built-in support for checking API availability, which ensures that you don’t accidentally use APIs that are unavailable on a given deployment target.

  - You use an *availability condition* in an if or guard statement to conditionally execute a block of code,

```
if #available(platform name version, ..., *){ //} else
{ //}
    if #available(iOS 10, macOS 10.12, *) {
        // Use iOS 10 APIs on iOS, and use macOS 10.12
APIs on macOS
    } else {
        // Fall back to earlier iOS and macOS APIs
}
```

Control Flow – Control Transfer Statements

- Early Exit - *guard*

- A *guard* statement, like an if statement, executes statements depending on the Boolean value of an expression.

- guard condition must be true in order for the code after the guard statement to be executed.

- Unlike an if statement, **a guard statement always has an else clause**—the code inside the else clause is executed if the condition is not true.

```
func greet(person: [String: String]) {
    guard let name = person["name"] else {
        return
    }
    print("Hello \(name)!")
    guard let location = person["location"] else {
        print("I hope the weather is nice near you.")
        return
    }
    print("I hope the weather is nice in \(location).")
}
greet(person: ["name": "John"])
// Prints "Hello John!"
// Prints "I hope the weather is nice near you."
greet(person: ["name": "Jane", "location": "Cupertino"])
// Prints "Hello Jane!"
// Prints "I hope the weather is nice in Cupertino."
```